Programming Assignment 1
Designing a web crawler

Jaka Kokošar, Danijel Maraž, and Toni Kocjan

Fakulteta za Računalništvo in Informatiko UL `dm9929@student.uni-lj.si`

Abstract. The article covers the work done in the scope of the first programming assignment as part of the subject web information extraction and retrieval.

Keywords: Data Extraction Retrieval Web Crawler Database Python · Second keyword · Another keyword.

1 Introduction

A web crawler is surely one of the most interesting pieces of software we can write. Mainly because it involves both the world wide web as well as efficient information retrieval and storing with the help of a back end database along with the robustness to handle anything the internet can throw at it. Because of this we decided to implement our crawler in Python which isn't the fastest language but is nice to read and not very verbose along with it's very simple database and web interaction libraries. The crawler like all others makes use of a frontier queue and a very standard Breadth First Search (BFS) strategy with multiple workers simultaneously accessing the frontier, adding pages and parsing them. For the page rendering we utilized the Python Selenium library which allowed us to interpret Javascript as most modern pages contain it. Naturally one of the most controversial aspects of a web crawler is whether the page owners impose any restrictions on which pages can be accessed and parsed. To tackle this problem we simply checked inside the robots.txt file which provided us with all the information we needed to know regarding the wishes of the site owners as well as a useful site map that gave us more URLs to add to the frontier.

2 General structure

Our main file *web_crawler.py* takes as an argument the number of desired worker threads. If no argument is given it defaults to 4. The list *sites* contains the four starting sites as given in the instructions as well as the others chosen by us at random but in consideration to how related they are to the first five. All of the sites are immediately put into our *frontier* queue. The starting arguments are then parsed and the individual worker threads initialized with the *ThreadPoolExecutor* and *submit_worker* functions. All workers are contained in the list *futures* which with the help of the function *wait* ensures that the program exits when none of the running workers are able to get a new URL from the *frontier* queue. The heart of our crawler is the class Worker which implements all the functionalities of one worker thread. In previous versions of our crawler workers were supported by a set of data structures however this has since been changed so that all of the data is stored directly in the database. This makes the crawling process considerably easier and more practical as it can be shut down and resumed at anytime on one or multiple machines.

2.1 Supporting data structures

– The *frontier* queue is the most important supporting data structure. It is implemented using the *Multiprocessing.queue()* class which is thread safe. Workers use it as the main source of new URLs to visit. It contains URLs in a basic canonized *string* form.
– The *visited_dict* dictionary contains URLs in a canonical *string* form that have passed all checks and will be visited in the future or have already been visited. It's main function is to assess whether our crawler has seen this URL before or not. It is implemented with the *multiprocessing.Manager()* dictionary class which is thread safe.
– The *site_domains* dictionary contains key value pairs of domain names and robot parser objects. We use it to check if we've already seen the root domain of the site we're taking from the *frontier*. In case we have we simply get our acquired robots parser object which has already been initialized on the site's robots.txt file. On the contrary if the root domain hasn't been seen before we map it to the corresponding robots parser object. It is implemented with the same class as *visited_dict*.
– The *documents_dict* dictionary contains hashes in *string* form of already parsed html documents. It serves as a very basic mechanism to keep our crawler from parsing sites which have a different URL but the very same contents. It is implemented with the same class as *visited_dict*.

2.2 Worker class

describe general flow here

Init

Call

Get_chrome_driver

Parse_robots

Parse_url

Fetch_url takes as it's arguments a url string representing the url of the site to be fetched, it's *id*, a boolean value is_binary and the appropriate robots file parser. Afterwards a request is sent to the url which depending on the outcome can raise an exception or not. If no exception is raised a series of if statements handle which type of response we're dealing with.

- We check if the url contains a file that we can download. This is done with our *should_download_and_save_file(url)* function. Alternatively we check the headers of the response if the file is of type *msword*, *powerpoint*, *wordprocessingml.document* or *presentationml.presentation*. In this case the *save_file* function is called with the url and response as arguments.
- If the boolean value (*is_binary*) is binary or if the content type is *image* the *save_image* function is ran with the url and response as arguments.
- If the response is of type *text\html* we attempt to render the page with the Selenium library. Since we have not figured out a practical way to make the library process already received responses we have to send another response to the url with the same library which then gets stored and the function *parse_page_content(site_id, url, status_code, datetime.datetime.now(), robots)* is ran. If we get an error while rendering we update the status of the page in our database to 500 as most likely the site does not allow us to render it with Selenium.
- If the content type is something else the url is removed from the database with the *remove_page(url, datetime.datetime.now())* function.

If our original request returned an error we get it's id from the database and update it's status to unreachable (404).

Parse_page_content takes as arguments the id of the site, it's url, the status code and the time of last access. It acquires the page source and makes a hash out of it and tries to find an existing page id if such a page is already in the database via the function *page_for_url(url)*. Afterwards it attempts to locate a duplicate page id with the function *page_for_hash(hashed)* by using the hash of the page contents.

- If a duplicate id is found with the hash and if the url already exists in our database we simply update our existing page entry with information about the duplicate.
- If a duplicate id is found with the hash but if the url does not exist in our database we insert a new duplicate entry with information about the duplicate.
- If no duplicate is found with the hash but if the url already exists in our database we simply update our existing page entry.

– If no duplicate is found with the hash and no url already exists in our database we insert a new page entry into the database.

After these initial steps we begin the process of parsing the page.

– Using the BeautifulSoup library we get all the tags that contain the attribute ”href” and check if they contain a valid government URL that hasn't already been visited. In such a case we add it to the frontier and update the pages status in the database.
– how we fetch buttons
– Images are collected in a similar fashion by searching for the *img* tag with BeautifulSoup. We then check if the links are valid and have not already been visited. In such a case we add them to the frontier however there have been cases when the links were invalid however the files could still be obtained by turning the links into their canonical form. A special few if statements are added at the end to handle such cases.

CODE SAMPLE

3 Database

4 Crawling statistics

5 Conclusion

References